Wrapping Operations for Atomicity and Durability

A Position Paper on How to Simplify NVM Programming for Extreme Performance

Ellis Giles[1]
erg@rice.edu

Kshitij Doshi[2]
kshitij.a.doshi@intel.com

Peter Varman[1]
pjv@rice.edu

[1]Electrical Engineering and Computer Science, Rice University
Houston TX, USA

[2]Software and Services Group, Intel
Chandler, AZ, USA

Problem Statement: The ability to access emerging high capacity storage class memories (SCM) [3] in fine-grained units is a tremendous advantage for the growing set of applications involving graph traversals, pointer chasing and columnar databases. SCM provides persistence at instruction granularity in stream with computing, providing new opportunities and challenges for structuring data-centric computation. Rethinking application design to benefit from a persistent memory tier [6] has the potential to deliver orders-of-magnitude gains as shown by in-memory data processing software such as VoltDB[1], SAP HANA[2], and IBM DB2 [8]. The use of SCM also enables energy-efficient handling of vast amounts of data that are accessed infrequently.

For application development requiring quick turnaround, this unification of storage and memory into a single directly-accessed persistent storage memory tier is a mixed blessing. It pushes upon developers the burden of ensuring that SCM stores are ordered correctly, flushed from caches, and if interrupted by a crash, do not leave objects in inconsistent states or cause corruption of metadata. While memory fences let a developer control the order in which stores from one CPU are made visible to others, the default memory store semantics supported by processors make no guarantees of *when* the value being written by a store instruction will actually be reflected in memory banks. Cache flush instructions guarantee update of the backing memory soon; but the writes are also done asynchronously from volatile store buffers, and so, maintain the risk of losing an update in the face of an uncontrolled machine restart. To address this gap, processor manufacturers will need to provide instructions to wait until a designated store has drained at the controller; we refer to such instructions generically as PSYNC (persistent sync) in this paper. How to use persistent syncs, memory fences, and cache flushes to ensure properly ordered, all-or-nothing updates of a sequence of stores to arbitrary addresses is a challenge that software designers will need to confront to benefit from the instruction-granular persistence of SCM.

This position paper advocates a software library approach to liberate programmers from the burden of explicit state management in SCM. Even if a programmer were to flush caches synchronously after each store in a train of correlated writes (by combining flushing with PSYNC), a machine failure that interrupts the sequence can still result in an unrecoverable system unless the state at which the sequence was interrupted is precisely knowable at the time of recovery. A journal or other metadata also in persistent memory needs to track the progress of persistent memory writes and make it available in order to effect recovery – in a manner similar to log-based recovery in databases [5]. A subtle but important aspect of any solution is the need to deal with the processor cache hierarchy. The extreme degree to which high performance applications are vulnerable to memory stalls when running on a modern superscalar CPU means that a good solution must exploit caches as efficiently as possible. However, since the cache subsystem independently evicts cache lines and writes their values to memory, these uncontrolled evictions complicate solutions by causing partial and out-of-order updates to SCM. When combined with the need to also flush metadata writes, it is difficult to maintain high performance when out-of-order execution is continuously impeded by flushes, PSYNCs, and memory fences.

SoftWrAP Approach. The SoftWrAP (for Software based Write Aside Persistence) approach we propose is a software derivative of an earlier hardware solution WrAP [4]. In WrAP [4] a controller on the SCM maintains a log of updates and handles spurious cache evictions with an overflow structure called a victim cache. The SoftWrAP approach provides a programmer with **wrap** library calls to demarcate a region of code (as shown Figure 1) from within which stores (called *wrapped* stores) are to be written to SCM in an all-or-nothing manner. When an SCM location X is updated for the first time it is associated with an address X' in DRAM via a hash map which redirects all wrapped accesses from X to X'. Cache eviction of X' does not change X, which is updated in a controlled way by the run time.

Programmer Annotated Atomic Region in SoftWrAP

```
// x, p, and p_malloc'ed array are persistent
   int wid = wrap_open
   begin
      x = 1;                    | wrap_store (wid, &x, 1);
      …
      p= p_malloc(100);         | temp = p_malloc(100);
                                | wrap_store(wid,&p,temp);
      …
      while(i<25)
      begin
         p[i] = i ;             | wrap_store(wid, p+ i, i);
        i++;
        ……
      end
   end
   wrap_close(wid);
```

Figure 1: SoftWrAP transformations (shown on the right)

Thus value communication takes place through the cache hierarchy while the record of updates is streamed to SCM asynchronously and concurrently in the form of redo log

This work is supported partially by an Intel SSG Research Grant

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

records. The programmer can reshape this value communication for a desired isolation level by the simple expedient of choosing between a location or its alias, and the programmer is equally free to perform non-atomic writes when non-determinism is tolerable (e.g., in recording performance profiles). When committed writes drain out of the log, aliasing may be discontinued.

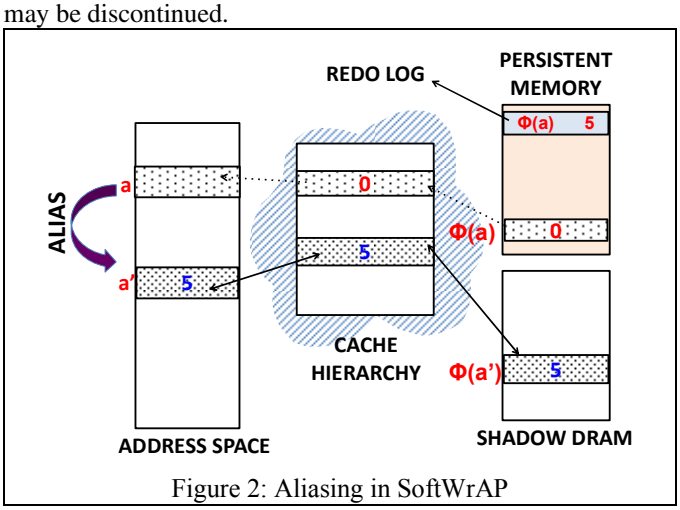

Figure 2: Aliasing in SoftWrAP

Figure 2 depicts the concept: initially **a** has the value 0 at its SCM location, φ(a). When it is accessed inside a wrap region, it is assigned a shadow location in DRAM with physical address φ(a') and virtual address a'. A wrapped write a=5 is treated as a write to the alias, a'=5, and is shown in the figure as a store in the cache with φ(a') as its backing location – thus any eviction from cache can only update φ(a') in DRAM. The redo log stores a copy of the new value as the record (φ(a), 5), and will be transferred to φ(a) as background activity. As a result of these redirections, cache evictions of modified values no longer affect locations in SCM, and the problem is reduced to that of ensuring that write ahead log in SCM is updated and flushed at the point a wrapped region ends (wrapClose).

A key benefit of this approach is that writes to the redo log can be streamed through non-temporal (non cache-allocating) writes. A second benefit is the avoidance of frequent cache flushes, PSYNC, and fence operations, since a single flushing write of the redo log at wrapClose achieves the necessary ordering. In our first implementation, wrap_load and wrap_store are implemented through compiler preprocessing, but in the future, a rich set of compiler optimizations can be brought to bear as read-only SCM locations can bypass wrap_load entirely, and intelligent register allocation can remove the need for frequent writes to the memory space for a persistent variable. To keep the size of the alias memory in check, the backing DRAM needs to be deallocated periodically by reclaiming entries for wraps that have closed and if the latest value of a variable has been copied from the corresponding redo log to its home location. The copying could also proceed from the alias table itself, to avoid having to read from the potentially slower, SCM based log entries. Proper sizing of the alias table lets one balance the retirement of log records (to free up the table memory) with the need for shadow DRAM space by new wraps.

Related Approaches: In comparison with an STM based approach such as in Mnemosyne [9], we let the developer choose the consistency and isolation model. This permits flexibility in the choice of an appropriate concurrency model particularly for a long running operation, instead of treating it as a monolithic STM transaction. An alternative to SoftWrAP is the use of an undo log approach but it would require that a store to each SCM variable b be preceded with a synchronous copy of the original value of b into an undo log record, and without the benefit of nontemporal streaming writes.

Evaluation: We examined graph data structure creation using the Graph 500 Benchmark [7]. We allocated the graph data structure in persistent memory using the **pmalloc** call in our API and manually wrapped every call to modify or read from memory addresses allocated in the persistent region. Creation time is shown in Figure 3. SoftWrAP is twice as fast for reliable graph construction and node insertions when compared to a Copy-On-Write Undo Log approach.

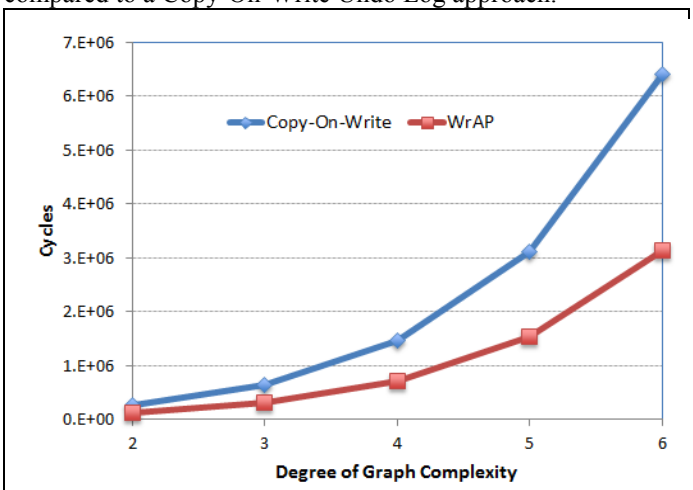

Figure 3: Graph Creation Time

REFERENCES

[1] VoltDB:. In http://voltdb.com..

[2] Farber, F., Cha, S.K., Primsch, J., Bornhovd, C., Sigg, S., and Lehner, W. *SAP HANA database: data management for modern business applications.* SIGMOD Rec. 40, 4 (Jan. 2012), 45–51.

[3] Freitas, R., and Wilcke, W. *Storage class memory, the next storage system technology*. IBM J. Res. and Dev., 52, 4/5 (2008).

[4] Giles, E., Doshi, K., and Varman, P. Bridging the programming gap between persistent and volatile memory using WrAP. Proc. ACM Computing Frontiers (2013).

[5] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17, 1 (Mar. 1992).

[6] Moraru, J., Andresen, D., Kaminsky, M., Binkert, N., Tolia, N., Munz, R., Ranganathan, P. *Persistent, protected and cached: Building blocks for main memory data stores*. In CMU Parallel Data Lab Trechnical Report, CMU-PDL-11-114 (Dec. 2011).

[7] Murphy, R.C., Wheeler, K.B., Barrett, B.W., and Ang, J.A., Introducing the Graph 500.

[8] Raman, V., et al *DB2 with BLU Acceleration: So Much More than Just a Column Store*, In Proc. VLDB 2013.

[9] Volos, H., Tack, A. J., and Swift, M. *Mnemosyne: Lightweight persistent memory*. Proc. ASPLOS (2011).